Windows
A Software Engineering Odyssey

Mark Lucovsky
Distinguished Engineer
Microsoft Corporation

Agenda

- History of NT
- Design Goals/Culture
- NT 3.1 Development vs. Windows 2000 Development
- Development for the next 10 years

NT Timeline first 15 years

- 11/88 Design Process Begins
- 2/89 First check in, coding Begins
- 7/93 NT 3.1 Ships
- 9/94 NT 3.5 Ships
- 5/95 NT 3.51 Ships
- 7/96 NT 4.0 Ships
- 12/99 NT 5.0 a.k.a. Windows 2000 ships
- 8/2001 Windows XP Ships
- 4/2003 Windows Server 2003 Ships

Unix Timeline first 20 years

- '69 Coding Begins
- '71 First Edition – PDP 11/20
- '73 Fourth Edition – Rewritten in C
- '75 Fifth Edition – Leaves Bell Labs, basis for BSD 1.x
- '79 Seventh Edition – One of the best
- '82 System III
- '84 4.2 BSD
- '89 SVR4 Unification of Xenix, BSD, System V
  - NT development begins

Linux Timeline first 12 years

- 8/91 Linus Announces he is working on Linux
- 9/91Linux version 0.01 released to web
- 6/93 Slackware, first commercial standalone distribution
- 3/94 Linux 1.0 released
- 3/95 Linux 1.2 released
- 6/96 Linux 2.0 released
- 1/98 Red Hat Advanced Development Labs (RHAD) founded
- 1/99 Linux 2.2.0 released
- 9/99 Red Hat (RHAT) Stock hits $135
- 12/99 VA Linux (LNUX) IPO open at $30, hits $300, closes at $250
- 1/01 Linux 2.4.0 released, stabilizes 9/01 as 2.4.10 with new VM implementation
- 8/02 Linux 2.4.19 released, 2.5.xx is in terrible shape lockups, fs corruption, etc.
- 7/03 Linux 2.6.TEST released
- 7/03 RHAT $6.75, LNUX $2.10

History of NT

- Team forms November 1988
- Six guys from DEC
- One guy from Microsoft
- Build from the ground up
  - Advanced PC Operating System
  - Designed for for desktops and servers
  - Secure, scalable SMP design
  - All new code
- Schedule: 18months (only missed our date by 3 years)

History of NT (cont.)

- Initial effort targeted at Intel i860 code-named N10, hence the name NT which doubled as N-Ten and New Technology
- Most development done on i860 simulator running on OS/2 1.2 (took about 30 minutes to boot)
- Microsoft built a single board i860 computer code named Dazzle including the supporting chipset and actually ran a full kernel, memory management, etc on the machine.
- Compiler came from Metaware with weekly UUCP updates sent to my Sun-4/200.
- Microsoft wrote a PE/Coff linker as well as a graphical cross debugger

Design Longevity

- OS Code has a long lifetime
- You have to base your OS on solid design principles
- You have to set goals, and not everything can be at the top of the list
- You have to design for evolution in hardware, usage patterns, etc.,
- Only way to succeed is base your design on a solid architectural foundation
- Development environments never get enough attention…

Goal Setting

- First job was to establish high level goals.
  - Portability
    - Ability to target more than one processor, avoid assembler, abstract away machine dependencies. We started the i386 port very late in order to avoid falling into a typical, Microsoft, x86 centric design.
  - Reliability
    - Nothing should be able to crash the OS.
      - Anything that crashes the OS is a bug.
      - Very radical thinking inside of Microsoft considering Win16 was cooperative multi-tasking in a single address space, and OS/2 had many similar attributes with respect to memory isolation
  - Extensibility
    - Ability to extend the OS over time
  - Compatibility
    - With DOS, OS/2, POSIX, or other popular runtimes.
    - This is the foundation work that allowed us to invent Win32 two years into NT OS/2 development.
  - Performance
    - All of the above are more important than raw speed!

NT OS/2 Design Workbook*

- Design of system captured in functional specs
- Written by engineers, for engineers
- Every functional interface was defined and reviewed
- Small teams can do this efficiently,
  - making this process scale is an almost impossible challenge
  - Senior developers are inundated with spec reviews and the value of their feedback becomes meaningless
  - You have to spread review duties broadly, and everyone must share the "culture"

* Now part of the Information Technology collection at the Smithsonian Institution's National Museum of American History

Developing a Culture

- To scale a development team, you need to establish a culture
  - Common way of evaluating designs, making tradeoffs, etc.
  - Common way of developing code and reacting to problems (build breaks, critical bugs, etc.)
  - Common way of establishing ownership of problems
- The Goal Setting exercise can be the foundation for the culture
- Keeping a culture alive as a team grows is a huge challenge
  - Windows original culture has unfortunately not survived…

The NT Culture

- Portability, Reliability, Security, and Extensibility ingrained as the teams top priority
  - Every decision was made in the context of these design goals
- Everyone owns all the code, so whenever something is busted anyone has a right and a duty to fix it
  - Works in small groups (< 150 people) where people cover for each other
  - Fails miserably in large groups
- Sloppiness is not tolerated, especially build/boot breaks
  - Great idea, but very difficult to nurture as group grows
  - Abuse and intimidation gets way out of control, can't keep calling people stupid and expect them to listen
- A successful culture has to accept that mistakes will happen and have a mechanism to recover

Development Environment

- NT 3.1 vs. Windows 2000
  - 10 years of growth
  - Development Teams
  - Source Code Control System
  - Mechanics of Writing Code
  - Team Size
  - Serialized Development
  - Defects

Development Team

- NT 3.1
  - Starts very small (6), grows very slowly to 200 people
  - NT Culture was commonly understood by all, practiced by most
- Windows 2000
  - Mass assimilation of other teams into the NT team
  - NT 4.0 had 800 developers, Windows 2000 had 1400
  - Original NT culture practiced by the old timers in the group, but keeping the culture alive was very difficult due to growth, physical separation, etc.
    - Diluted culture leads to much conflict
      - Accountability: I don't "own" the code that is busted, see Mark., maybe he will fix it
      - reliability vs. new features vs. performance at any cost
      - 64-bit portability vs. new features available in 32bit modes only

Source Code Control System (NT 3.1)

- Internally developed, maintained by a non-NT tools team
  - No branch capability, but with small team, it was not needed
- 10-12 well isolated source "projects", 6M LOC
- Informal project separation worked well
  - minimal obscure source level dependencies
- Small hard drive (120-300mb) could easily hold entire source tree
- Developer could easily stay in sync with changes made to the system

Source Code Control System (Windows 2000)

- Windows team takes ownership of source code control system which at this point is on life support
- Branch capability sorely needed, tree copies used as substitute, so merging is a nightmare
- 180 source "projects" 29M LOC
- No project separation rules
- Reaching "up and over" was very common as developers tried to minimize what they had to carry on their machines to get their jobs done
- Full source base required about 50Gb of disk space
- To keep a machine in synch was a huge chore (1 week to setup, 2 hours per-day to synchronize)
- BV went 0 for 30 trying to build NT in 2/99

Mechanics of Writing Code (NT 3.1)

- Safe synch period in effect for ~4 hours each day, all other times the rule is check-in when ready
- Build lab synchs during morning safe synch period, and starts a complete build.
  - Build breaks are corrected manually during the build process (1-2 breaks was normal)
- Complete build time is 5 hours on ~486/50
- Build is boot tested with some very minimal testing before release to stress testing
  - Defects corrected with incremental build fixes
- 4pm, stress testing on ~100 machines begins

Mechanics of Writing Code (Windows 2000)

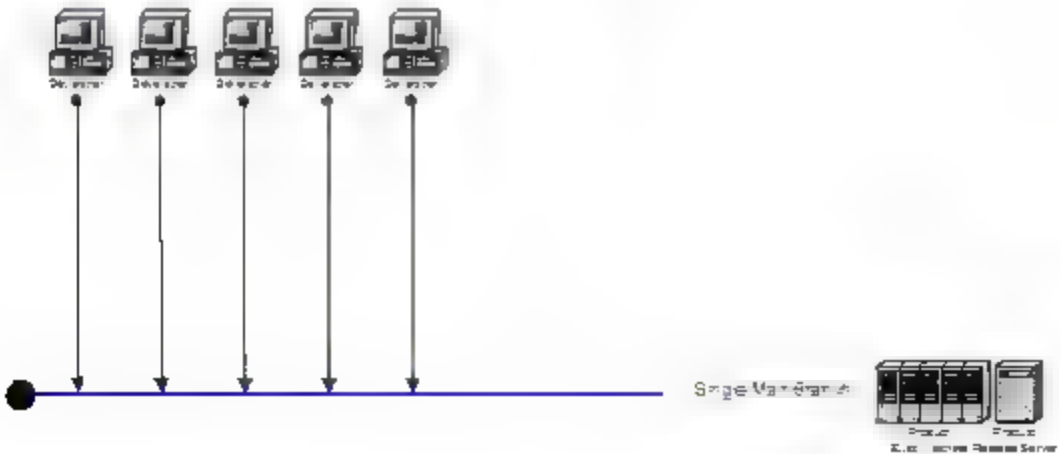
- **Developers are not allowed to change the source tree without explicit, email/written permission (Electronic White Board or EWB system)**
  - Build lab manually approves each check-in using a combination of email, web, and bug tracking database
- **Build lab approves about 100 changes each day and manually issues the appropriate synch and build commands**
  - Build breaks are corrected manually, and when they occur, all further build processing is halted
  - A developer that mistypes a build instruction can stop the build lab, which in turn stops over 5 000 people
- **Complete build time is 8 hours on 4 way PIII Xeon 550 with 50Gb disk and 512k RAM.**
  - BUT complete builds are rarely done. management is scared of the code!
- **Build is boot tested and assuming we get a boot, extensive baseline testing begins**
  - Testing is a semi-manual, semi-automated process, lots of room for errors
  - Defects occurring in the boot or test phase must be corrected before build is "released" for stress testing
- **4pm, stress testing on ~1000 machines begins**

Team Size

Product	Dev Team Size	Test Team Size
NT 3.1	200	140
NT 3.5	300	230
NT 3.51	450	325
NT 4.0	800	700
Win2k	1400	1700
Today	2800	2100

Serialized Development

- The model from NT 3.1 -> Windows 2000
- All developers on team check-in to a single main line branch
- Master build lab manually synchs to main branch and builds and releases from that branch
- Checked in defect affects everyone waiting for results



Defect Rates and Serialization

- Compile time or run time bugs that occur in a developers office only affect that developer
- Once a defect is checked-in, the number of people affected by the defect increases
- Best developers are going to check-in a runtime or compile time mistake at least twice each year
- Best developers will be able to cope with a checked-in compile time or run time break very quickly
  - about 20 minutes end-to-end, once they have been contacted
- As the code base gets larger, and as the team gets larger, these numbers typically double

Defect Rates Data

- With serialized development:
  - Good, small teams operate efficiently
  - Even the absolute best large teams are always broken, and always serialized

Product And Dev Team Size	Defects: Per year Per Dev	Time to Fix: Per Defect	Defects: Per Day	Total Defect Fix Time
NT 3.1, 200	2	20 minutes	1	20 minutes
NT 3 5, 300	2	25 minutes	1 6	41 minutes
NT 3 51, 450	2	30 minutes	2 5	1 2 hours
NT 4.0, 800	3	35 minutes	6 6	3 8 hours
Win2k, 1400	4	40 minutes	**15.3**	**10.2 hours**

Development Environment Summary

- NT 3.1
  - Fast and loose development, lots of fun, lots of energy
  - Few barriers to getting your work done
  - Defects serialized parts of the process, but didn't stop the whole machine, minimal down time
- Windows 2000
  - Source code control system bursting at the seams
    - Staying in sync, doing your work, nearly impossible
  - Excessive process management serialized the entire development process
    - 1 defect stops 1400 devs, 5000 team members!
  - Management's main lever was to slow down the dev process by reducing the check in rate making devs even less productive/happy
  - Resources required to build a complete instance of NT were excessive giving few developers a way to be successful

Development for the next 10 years

- Procedural Changes
  - GOAL: Get control over what makes it into a product, why we are doing it, how we make decisions
  - What process is used to establish product feature set
    - Can Shared Goals and Culture help guide us?
    - Are releases feature driven? Date drive? Both?
    - Do we dream about features and then bet on them before there is a sound technical design?
  - Design and implement feature definition process
    - What is a feature specification?
    - Is there a feature removal contingency plan? Should spec design this in?
    - Do features need to pass a minimum technical bar before they can be considered for product inclusion?
- Mechanical Changes
  - GOAL: Bring back the good old days where developers feel like they can make rapid progress, have fun writing and debugging code, feel accountable for their code and for the code written by their team
  - Address Source Code Control System
  - Address Source Code Layout to formalize dependencies, allow partitioned builds
  - Make the large team work like a set of small teams
    - Windows is already organized into reasonable size development teams
    - Goal is to allow these teams to work as a team when contributing source code changes rather than as a group of individuals that happen to work for the same VP
    - Pursue Parallel Development Team Level Independence
  - Automate the build process, enable multi-machine builds
- Mechanical Changes were pursued for Windows XP and beyond,
- Procedural Changes were abandoned to be revisited later if necessary, WELT is currently investing in this area for Longhorn and beyond

Source Code Control System

- New source code control system identified 3/99 (SourceDepot)
- Native branch support
- Scalable, high speed, client server architecture
- New machine setup 3 hours vs. 1 week
- Normal synch 5 minutes vs. 2+ hours

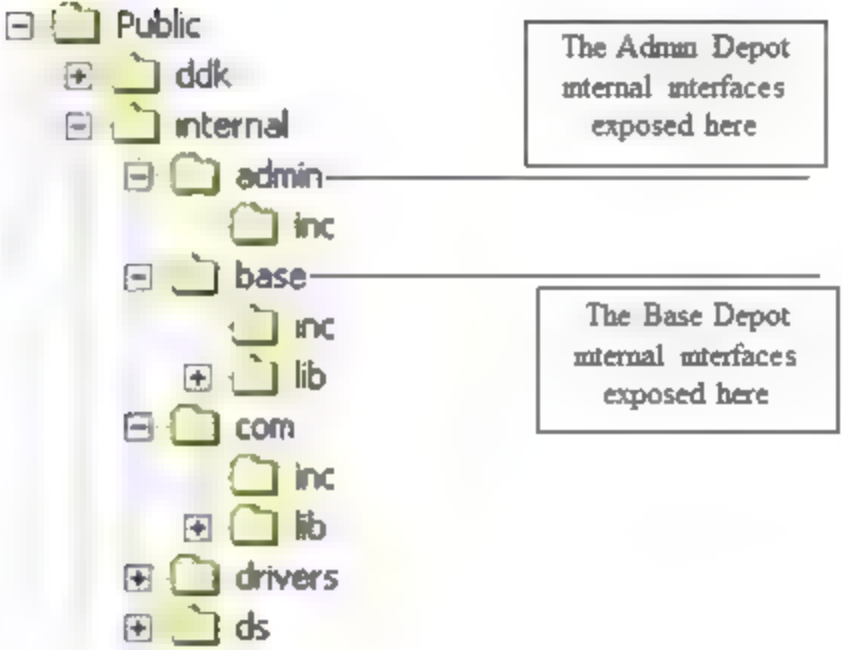
Source Code Control System (cont.)

- Transition to SourceDepot done on LIVE Win2k code base
- Hand built SLM -> SourceDepot migration system allowed us to keep in synch with the old system while transitioning to SourceDepot and changing the source code layout
- SLM Shutdown, SourceDepot enabled DURING Win2k RTM Party

Source Code Restructuring

- 16 Depots for covering each major area of source code
  - Vs 190+ SLM projects
- Organization is focused on:
  - minimizing cross depot dependencies to reduce defect rate
  - Sizing depots to compile in a reasonable amount of time (30-60min)
  - To build a depot, all you need is the code for that project, AND the public/root project
  - Cross project sharing is explicit

Explicit Internal Interface Sharing



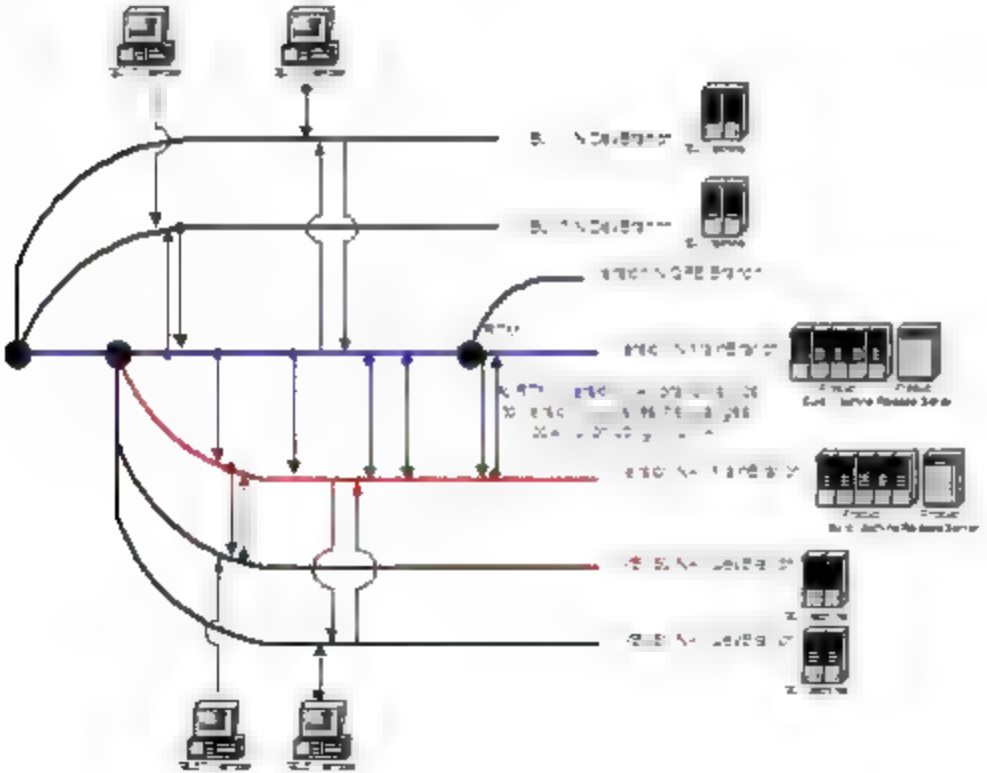
Team Level Independence

- Each team determines its own check-in policy
  - Enable rapid, frequent check ins
- Teams are isolated from mistakes made by other teams
  - When errors occur, only the team causing the error is affected
  - A build boot, or test break only affects a small subset of the product group
- Each team has:
  - Their own view of the source tree
  - Their own mini build lab
  - Builds and tests (mini BVT) an entire installable build
- Any developer with *adequate resources** can easily duplicate a mini build lab
  - build and release a completely installable Windows System

**adequate resources*: 2x2.4Ghz = 14 hours, Distributed, multi-machine (4) build $10k == 6 hours.

Parallel Development

Enabled by “Team Level Independence”



Automated Builds

- Team level build labs (VBL) can run 100% hands off
  - 10am and 10pm full sync and full build
    - Build failures are auto detected and mailed to the team
    - Successful builds are automatically released with automatic notification to the team
- Each VBL can build:
  - 4 platforms (x86 fre/chk, ia64 fre/chk) = 8 builds each day, 56 each week
  - No manual steps at all. 100% Hands off automatic
  - 7 VBLs in Win2k Group
  - Majority of builds work*, but failures when they occur are isolated to a single team

Productivity Gains

- Developers can easily switch from working on release N to release N+1
- Developers in one team will not be impacted by mistakes/changes made by other teams
- Developers have long, frequent checkin windows (Base team has a 24x7 open checkin window, vs. 2-3 hour per day checkin window with manual approval used during W2K
- Source code control system is fast and reliable
- Testing is done on complete builds instead of assorted collections of private binaries
  - What is in the source code control system is what is tested

How is it working? – Initial Assessment

- Source code control system is working very well
  - No scaling problems, easily handling 5100 total user enlistments and 411,000 files
  - Today 515k files 10k active users, 18781 total
- Source code restructuring is working well
  - No new depots added, explicit sharing between projects still the rule
- Parallel Development is working very well
  - Teams feel independent and able to control their own destiny
  - Per-team serialization only occurs when a team "reverse integrates" their changes into the main branch

How is it working? – 3 years Later

- Huge productivity gains over Win2k builds
  - Number of checkins per build
  - Number of bugs fixed per build
- Daily MAIN builds are of higher quality
- Debilitating breaks are isolated in VBLs instead of MAIN, affecting a subset of developer teams
- RI (team branch into main) frequency has diminished over time. Factors:
  - Number of devs checking in per VBL
    - XP/WS03 Split forced the bulk of the dev team into a single branch
  - Time required to build/BVT
    - VBL Hardware and staffing directly impacts this
      - Single machine build+BVT 18 hours
      - Distributed Build + BVT ~ 10 hours
  - Ability to lock down the RI branch to produce a great set of bits to push into main

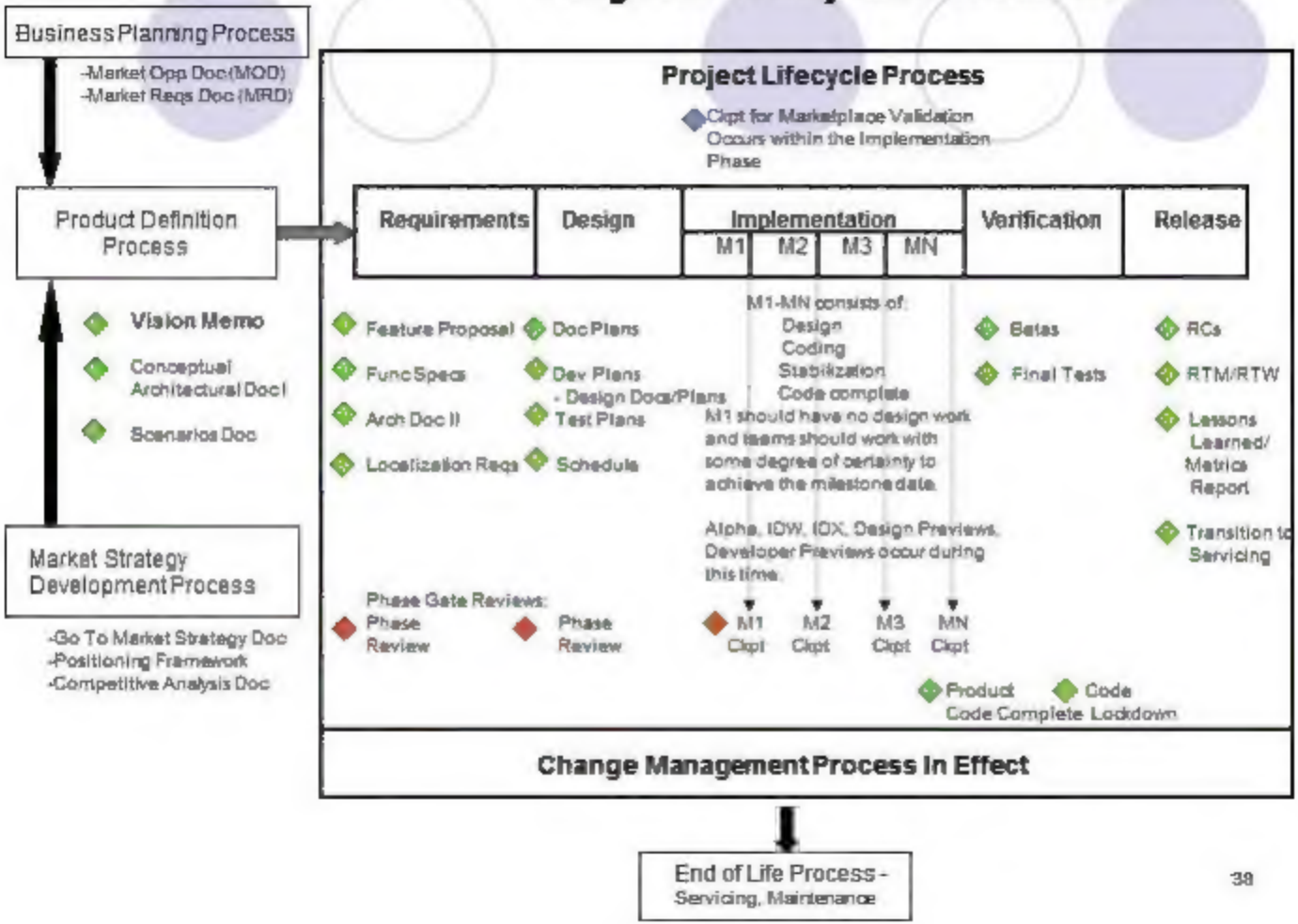
Points of Pain

- Cross branch collaboration
  - This doesn't work and never will
  - To collaborate, you need to be in the same branch
  - Current branch architecture is optimized for VP level independence. NOT VP level collaboration
- Uniform VBL Metrics and reporting
  - How do we track progress without uniform statistics
- BVT Uniformity
  - Hardware, topology, general automation
- Auto-magic RI coordination
  - Sloppy Understaffed VBL that do not run BVTs before RI, slip in last minute check-ins, etc. can still fake it and hose the system
- Size of VBL/Branch dev team directly impacts branch quality and ability to RI
  - Need to constantly monitor this and adjust
- Need to invest in "Procedural Changes"

Procedural Changes

- Re-emphasize, and re-vitalize Architecture process
  - Shared definition and reason for caring
  - Ingrained in culture
    - Respect layering and boundaries, fear complexity and dependencies
- Formalize Customer Connections
  - Experience the system as a customer would
  - Use the same diagnosis/support tools as PSS
- Quality through Tools
  - Watson, PREfix/PREfast, Test Automation, Driver Verifier, App Verifier, Fault Injection, etc.
- Formalize Product Development Process
  - See next slide
- Engineering Excellence (code & process quality)
  - Ingrain these principles in the culture of the teams
  - Do the right thing for all the right reasons

Project Lifecycle Framework



Summary

- The initial NT development environment and culture worked well for the first few years
- Ten years of team and code growth forced a major re-design of the development environment and culture
- With the new environment in place, the team is working a lot like they did in the NT 3.1 days with a small, fast moving, development team
- Mechanical Changes can solve the mechanical problems, but can't compensate for poor procedural processes

Questions